

Floyd-Hoare Style Program Verification (FMSE Course)

Deepak D'Souza

Department of Computer Science and Automation
Indian Institute of Science, Bangalore.

24 Feb 2026

Outline of these lectures

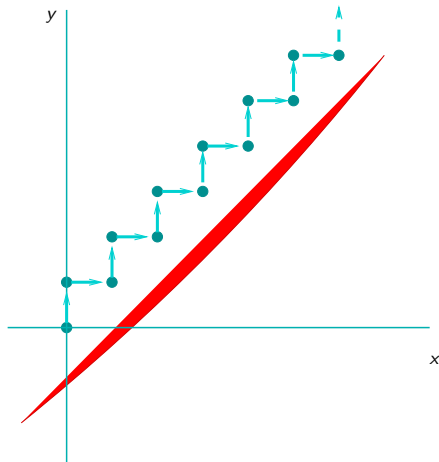
- 1 Overview
- 2 Hoare Triples
- 3 Proving assertions
- 4 Inductive Annotation
- 5 Nested Loops
- 6 Function Contracts
- 7 VCC

Checking Pre/Post Assertions in Programs

- Moving on from reasoning about **models** to reasoning about **code**.
- Still a **deductive** style of verification.
- Helps us to verify assertions and also refinement-based functionality verification.

Example Program and Property

```
x := 0;
y := 0;
while (*) {
  if (x < y)
    x++;
  else
    y++;
}
// assert y != x - 1
```



How would one check that this program satisfies the given assertion?

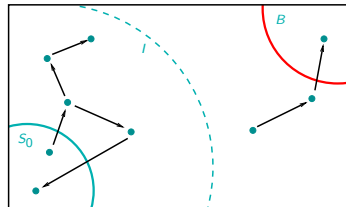
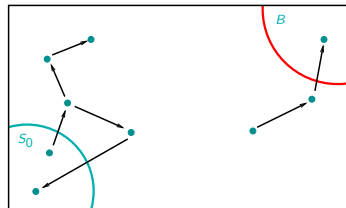
Idea of Deductive Verification

Problem: Given a transition system $\mathcal{T} = (S, S_0, \rightarrow)$ and an set of unsafe states $B \subseteq S$, does an execution of $\mathcal{T}$ reach a state in B ?

Find a set of states I such that

- ❶ $S_0 \subseteq I$ (initial states belong to I)
- ❷ $s \in I$ and $s \rightarrow s'$, implies $s' \in I$ (I is inductive wrt transitions)
- ❸ $I \cap B = \emptyset$ (I disjoint from Bad states).

Such an I is called an **adequate inductive invariant**.



Idea of deductive verification

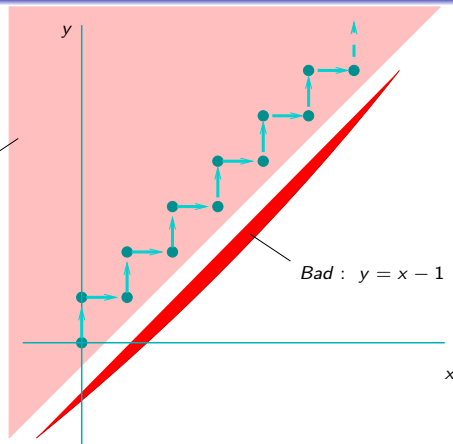
```
x := 0;
y := 0;
while (*) {
  if (x < y)
    x++;
  else
    y++;
}
// assert y != x - 1
```

$I : x \leq y$

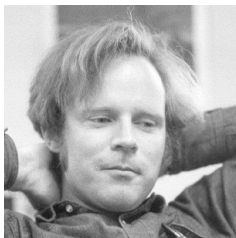
$Bad : y = x - 1$

I is an adequate inductive invariant:

- 1 $s_0 \in I$ (initial state belongs to I)
- 2 $s \in I$ and $s \rightarrow s'$, implies $s' \in I$ (I is inductive wrt trans)
- 3 $I \cap B = \emptyset$ (I disjoint from Bad states).



Floyd-Hoare Style of Program Verification



Robert W. Floyd: “Assigning meanings to programs” *Proceedings of the American Mathematical Society Symposia on Applied Mathematics* (1967)



C A R Hoare: “An axiomatic basis for computer programming”, *Communications of the ACM* (1969).

Floyd-Hoare Logic

- A way of asserting properties of programs.
- Hoare triple: $\{A\}P\{B\}$ asserts that “Whenever program P is started in a state satisfying condition A , if it terminates, it will terminate in a state satisfying condition B .”
- Example assertion: $\{n \geq 0\} P \{a = n + m\}$, where P is the program:

```
int a := m;
int x := 0;
while (x < n) {
  a := a + 1;
  x := x + 1;
}
```

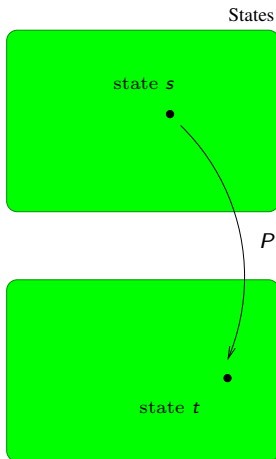
- Inductive Annotation (“consistent interpretation”) (due to Floyd)
- A proof system (due to Hoare) for proving such assertions.
- A way of reasoning about such assertions using the notion of “Weakest Preconditions” (due to Dijkstra).

A Simple Programming Language

- `skip` (do nothing)
- `x := e` (assignment)
- `if b then S else T` (if-then-else)
- `while b do S` (while loop)
- `S ; T` (sequencing)

Programs as State Transformers

- Program state is a valuation to variables of the program:
 $States = Var \rightarrow \mathbb{Z}$.
- View program P as a **partial** map $\llbracket P \rrbracket : States \rightarrow States$.



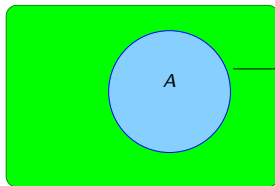
$s : \langle x \mapsto 2, y \mapsto 10, z \mapsto 3 \rangle$

```
y := y + 1;  
z := x + y
```

$t : \langle x \mapsto 2, y \mapsto 11, z \mapsto 13 \rangle$

Predicates on States

All States



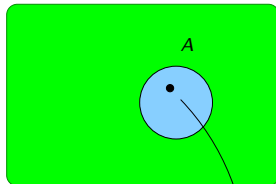
States satisfying
Predicate A

Eg. $0 \leq x \wedge x < y$

Assertion of “Partial Correctness” $\{A\}P\{B\}$

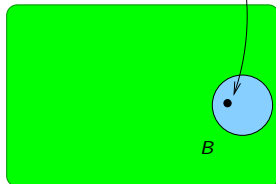
$\{A\}P\{B\}$ asserts that “Whenever program P is started in a state satisfying condition A , either it will not terminate, or it will terminate in a state satisfying condition B .”

All States



$$\{10 \leq y\}$$

P



$y := y + 1;$
 $z := x + y$

$$\{x < z\}$$

Mathematical meaning of a Hoare triple

- View program P as a **relation** on States (allows non-termination as well as non-determinism)

$$\llbracket P \rrbracket \subseteq \text{States} \times \text{States}.$$

Here $(s, t) \in \llbracket P \rrbracket$ iff it is possible to start P in the state s and terminate in state t .

- $\llbracket P \rrbracket$ is possibly non-deterministic, in case we also want to model non-deterministic assignment etc.
- Then the Hoare triple $\{A\} P \{B\}$ is true iff for all states s and t : whenever $s \models A$ and $(s, t) \in \llbracket P \rrbracket$, then $t \models B$.
- In other words $\text{Post}_{\llbracket P \rrbracket}(\llbracket A \rrbracket) \subseteq \llbracket B \rrbracket$.

Example programs and pre/post conditions

```
// Pre: true
```

```
if (a <= b)
```

```
    min := a;
```

```
else
```

```
    min := b;
```

```
// Post: min <= a && min <= b
```

```
// Pre: 0 <= n
```

```
int a := m;
```

```
int x := 0;
```

```
while (x < n) {
```

```
    a := a + 1;
```

```
    x := x + 1;
```

```
}
```

```
// Post: a = m + n
```

Example programs and pre/post conditions

```
// Pre: true
```

```
if (a <= b)
```

```
    min := a;
```

```
else
```

```
    min := b;
```

```
// Post: min <= a && min <= b
```

```
// Pre: 0 <= n
```

```
int a := m;
```

```
int x := 0;
```

```
while (x < n) {
```

```
    a := a + 1;
```

```
    x := x + 1;
```

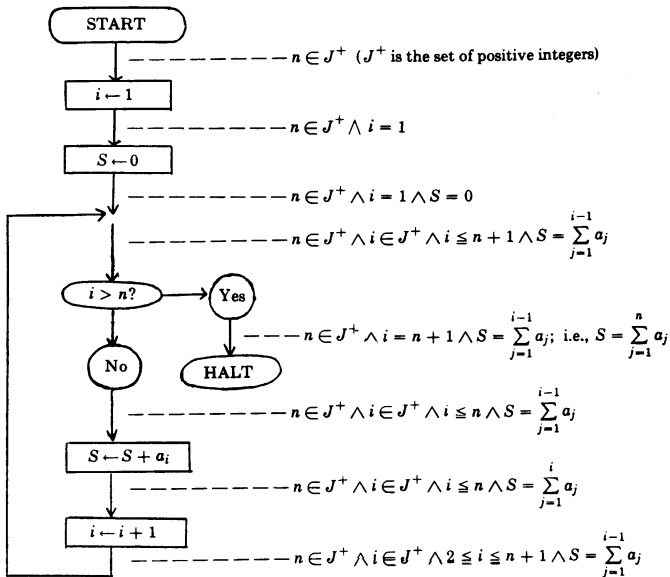
```
}
```

```
// Post: a = m + n
```

Exercise:

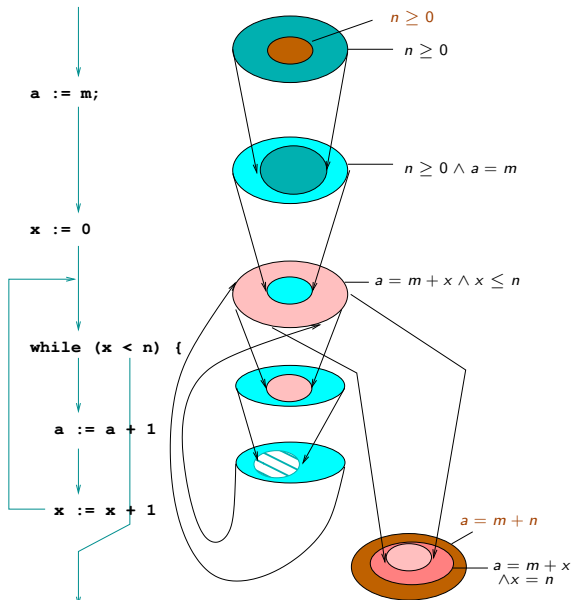
- 1 Give A , P , B for which $\{A\} P \{B\}$ *does not* hold.
- 2 How will you phrase the assertion checking problem in motivating program as a pre-post checking problem?

Floyd style proof: Inductive Annotation



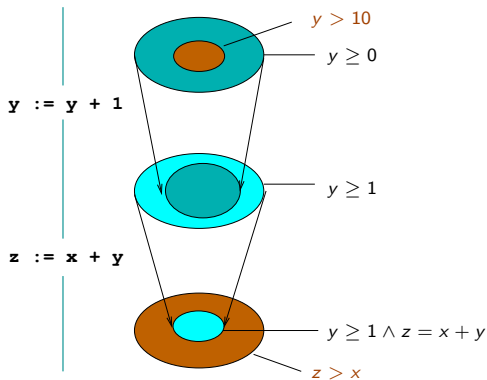
Inductive annotation based proof of a pre/post specification

- Annotate each program point i with a predicate A_i
- Successive annotations must be **inductive**:
 $\llbracket S_i \rrbracket(\llbracket A_i \rrbracket) \subseteq \llbracket A_{i+1} \rrbracket$,
OR logically:
 $A_i \wedge [S_i] \implies A'_{i+1}$.
- Annotation is **adequate**:
 $Pre \implies A_1$ and
 $A_n \implies Post$.
- Adequate annotation constitutes a proof of $\{Pre\} \text{ Prog } \{Post\}$.



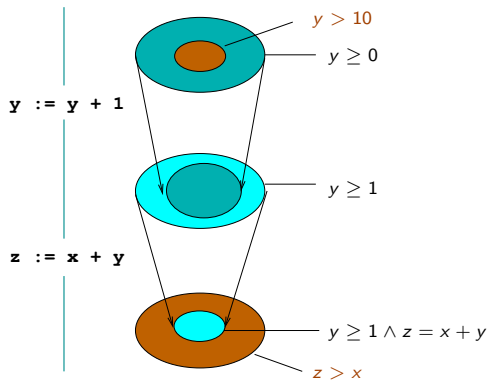
Example of inductive annotation

To prove: $\{y > 10\} y := y+1; z := x+y \{z > x\}$



Example of inductive annotation

To prove: $\{y > 10\} \ y := y+1; \ z := x+y \ \{z > x\}$



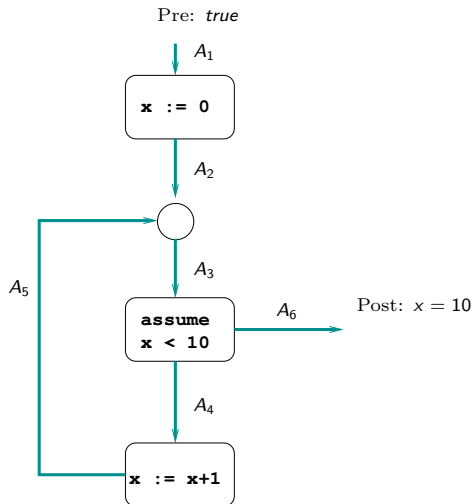
Logical proof obligations (VCs):

$$\begin{aligned}
 & (y > 10 \implies y \geq 0) \wedge ((y \geq 1 \wedge z = x + y) \implies z > x) \wedge \\
 & ((y \geq 0 \wedge y' = y + 1 \wedge x' = x \wedge z' = z) \implies y' \geq 1) \wedge \\
 & ((y \geq 1 \wedge z' = x + y \wedge x' = x \wedge y' = y) \implies y' \geq 1 \wedge z' = x' + y')
 \end{aligned}$$

Exercise

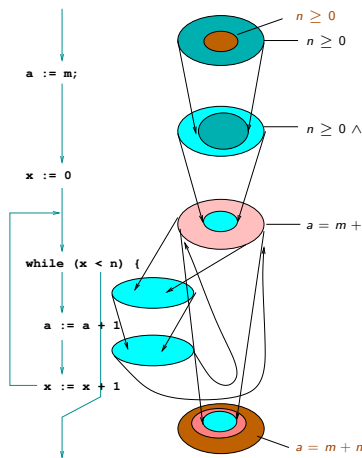
Prove using Floyd-style annotation:

```
// Pre: true
int x := 0;
while (x < 10)
  x := x + 1;
// Post: x = 10
```



Also write out the proof obligations (verification conditions).

Adequate loop invariant



An **adequate** loop invariant needs to satisfy:

- $\{n \geq 0\} \text{ a := m; x := 0 } \{a = m + x \wedge x \leq n\}.$
- $\{a = m + x \wedge x \leq n \wedge x < n\} \text{ a := a+1; x := x+1 } \{a = m + x \wedge x \leq n\}.$
- $\{a = m + x \wedge x \leq n \wedge x \geq n\} \text{ skip } \{a = m + n\}.$

Verification conditions are generated accordingly.

Note that $a = m + x$ is **not** an adequate loop invariant.

Generating Verification Conditions for a program

assume Pre

S_1

while (b) { ← *invariant Inv*

S_2

}

S_3

assert Post

The following VCs are generated:

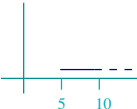
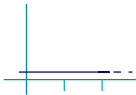
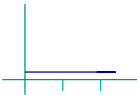
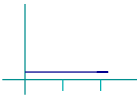
- $Pre \wedge [S_1] \implies Inv'$
Or: $Pre \implies WP(S_1, Inv)$
- $Inv \wedge b \wedge [S_2] \implies Inv'$
Or: $(Inv \wedge b) \implies WP(S_2, Inv)$
- $Inv \wedge \neg b \wedge [S_3] \implies Post'$
Or: $Inv \wedge \neg b \implies WP(S_3, Post)$

Adequate loop invariant

What is a “good” loop invariant for this program?

```
x := 0;
while (x < 10) {
  if (x >= 0)
    x := x + 1;
  else
    x := x - 1;
}
assert (x <= 11);
```

Adequate loop invariant

	<i>Canonical Invariant</i>	<i>Not-inv</i>	<i>Inv,not-ind</i>	<i>Inv,ind,not-adeq</i>	<i>Inv,ind,adeq</i>
<pre> x := 0; while (x < 10) { if (x >= 0) x := x + 1; else x := x - 1; } assert(x <= 11); </pre>	$0 \leq x \leq 10$ 	$5 \leq x$ 	$-1 \leq x$ 	$0 \leq x \leq 12$ 	$0 \leq x \leq 11$ 

Handling nested loops

Verification conditions generated:

- $Pre \wedge [S_1] \implies Inv1'$
- $Inv1 \wedge b \wedge [S_2] \implies Inv2'$
- $Inv2 \wedge c \wedge [S_3] \implies Inv2'$
- $Inv2 \wedge \neg c \wedge [S_4] \implies Inv1'$
- $Inv1 \wedge \neg b \wedge [S_5] \implies Post'$

assume Pre

S_1

while (b) {

inv1

S_2

while (c) {

inv2

S_3

}

S_4

}

S_5

assert Post

Contracts for Recursive Functions

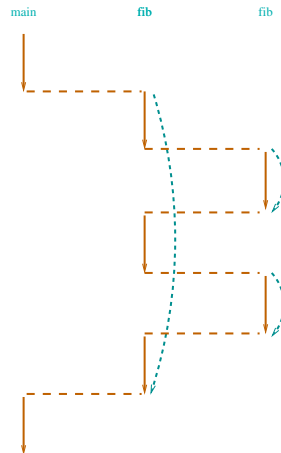
```
main() {  
    result = fib(5);  
    assert (result > 2);  
}  
  
// requires x >= 0  
// ensures (result >= x) && (result > 0)  
int fib(int x) {  
    if (x < 2)  
        return 1;  
    else  
        return fib(x-1) + fib(x-2);  
}
```

Proof obligations generated:

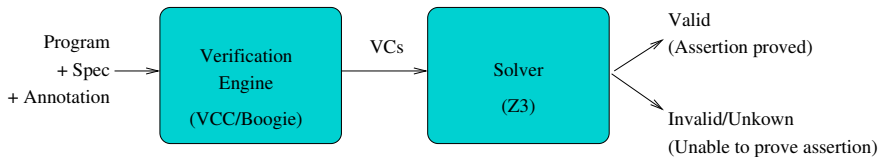
- At every call to `foo()`, the **requires** of `foo` holds.
- Assuming the **requires** of `foo` and the **ensures** of each function `bar` called in `foo`, `foo` satisfies its **ensures**.

Soundness of Function Contracts

- Consider an execution of a program with valid contract annotation



How VCC Works (in Principle)



Conclusion

- Features of this Floyd-Hoare style of verification:
 - Tries to find a proof in the form of an **inductive annotation**.
 - A Floyd-style proof can be used to obtain a Hoare-style proof; and vice-versa.
 - Reduces verification (given key annotations) to checking satisfiability of a logical formula (VCs).
 - Is flexible about predicates, logic used (for example can add quantifiers to reason about arrays).
- Main challenge is the need for user annotation (adequate loop invariants).
- Can be increasingly automated (using learning techniques).